CS2106 Cheatsheet

github.com/reidenong/cheatsheets, AY24/25 Sem 1

Introduction to Operating Systems

Ways to structure an OS

Monolithic

- Kernel is one big special program comprising all OS services
- Good engineering principles are possible through modularization and separation of interfaces and implementation
- Advantages are: well understood, good performance
- Disadvantages are: very complicated internal structure

Microkernel

- Kernel is very small and clean, and only provides very basic and essential facilities
- Higher level OS services are built on top of the basic facilities and run outside of the kernel, using IPC to communicate
- Advantages are: kernel is more robust, extensible. Better isolation and protection
- Disadvantages are: Lower performance

Virtual Machines

VMs provide a virtualization of underlying hardware, which allows OSes to run on top of the VM as an illusion of hardware. This is created and managed by Hypervisor.

Type 1 Hypervisor

- Provides individual virtual machines to guest OSes

Type 2 Hypervisor

- Provides virtual machines to guest OSes on top of a host OS

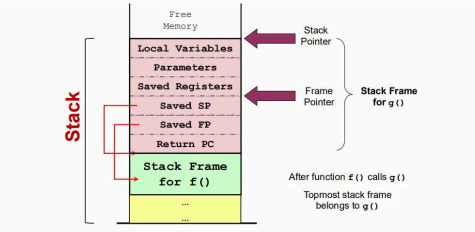
Process Abstraction

Function Calls

Stack Memory

Function calls have control issues, like needing to jump to the function body and then returning to caller, as well as passing parameters and returning results. We use a stack.

The top of a stack region is indicator by the stack pointer `sp`. The stack grows downwards, ie. towards lower memory addresses.



Executing a function call

Caller setup

Caller needs to plan space to reserved

- 8 bytes to save `$sp` and `$fp`
- 4 * `n` bytes for arguments
- 4 bytes for `$ra`

1. Save `$fp` and `$sp` to stack
2. Reserve space as planned above
3. Save arguments to stack
4. Call `f()`

Callee execution

1. Save `$ra` to stack
2. Save registers we want to use to stack
3. Do work
4. Write result to stack frame
5. Restore registers
6. Restore `$ra`
7. Return

Caller teardown

1. Save result
2. Restore `$sp` and `$fp`

Sample MIPS program

```
f:
sw   $ra, 16($fp)    # save $ra$
addi $sp, $sp, 8     # make space 2 save regs
sw   $t0, 20($fp)    # save old_register_1
sw   $t1, 24($fp)    # save old_register_2
lw   $t0, 8($fp)     # load arg_1
lw   $t1, 12($fp)    # load arg_2
add  $t1, $t0, $t1   # do function work
sw   $t1, 8($fp)     # save result @ arg_1
lw   $t0, 20($fp)    # restore old_register_1
lw   $t1, 24($fp)    # restore old_register_2
addi $sp, $sp, -8    # del save register space
lw   $ra, 16($fp)    # restore $ra
jr   $ra             # return to caller

main:
sw   $fp, 0($sp)     # add fp to stack
mov  $fp, $sp        # copy $sp to fp
sw   $sp, 4($sp)     # add sp on top of stack
addi $sp, $sp, 20    # reserve 20B of space
la   $t0, a          # load address of a
lw   $t0, 0($t0)     # load value of a into a
sw   $t0, 8($fp)     # store a into arg_1
la   $t0, b          # load address of b
lw   $t0, 0($t0)     # load value of b
sw   $t0, 12($fp)    # store b into arg_2
jal  f               # function call
lw   $t0, 8($fp)     # store result into $t0
la   $t1, y          # get address of y
sw   $t0, 0($t1)     # load val into y
lw   $sp, 4($fp)     # restore $sp
lw   $fp, 0($fp)     # restore $fp
li   $v0, 10         # system call
syscall
```

Dynamic Memory

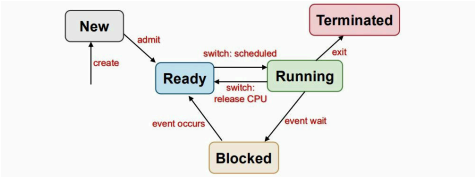
- Allocated only during runtime, cannot be in data
- No definite deallocation timing, cannot be in stack
- Memory is allocated in the Heap

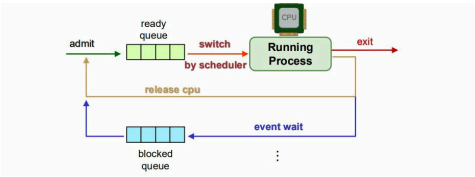
Processes

- A process is a program in execution. The OS provides an abstraction that creates an illusion of a single process running on a single CPU. The OS also protects the execution context of each process from others.

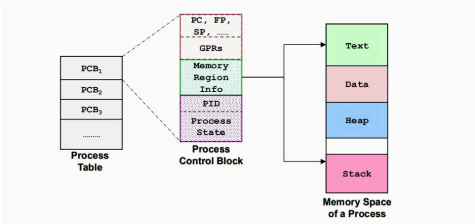
Memory Context	Hardware Context	OS Context
• Text • Data • Stack • Heap	• Registers • PC • `$sp` • `$fp`	• PID • PPID • Resources used

5 State process model





Process Control Block (PCB) & Table



System Calls

General Mechanism

1. User program calls a library function
2. Library call places system call number in a register
3. Library call switches to kernel mode (TRAP instruction)
4. Kernel determines system call handler
5. Kernel executes system call handler
6. Kernel returns to user mode
7. Library function returns to user program

Exceptions and Interrupts

Exceptions are synchronous and occur due to program execution. Interrupts are asynchronous and occur due to external events. Both result in automatic transfer of control to the handler. The handler saves the context (CPU state, Registers) and executes the handler code.

Process abstraction in Unix

Process Creation

`fork()` creates a duplicate of the current executable image, with memory as a copy of the parents (copy-on-write). This means both the child and parent share the same memory until one writes to it, whereupon the page will be changed.

`execl()` replaces the current executing process image with a new one.

```
execl(char *path, char *arg0, char *argn, NULL);
```

The `init` process is the first process created by the kernel at boot time, and is the root process.

Process Termination

`wait(int *status)` waits for a child process to terminate, and returns the PID of the terminated child. The status is stored in the status variable.

- `wait()` is blocking. The call cleans up remainder of child system resources, as well as killing zombie processes. Other variants include waitpid() (wait for specific child) and waitid() (wait for any child to change status).
- `exit()` creates zombie processes. Process info cannot be deleted until parent calls `wait()` in case information is needed. zombies cannot be killed.

Zombies and Orphans

Orphans happen when the parent process terminates before child process. `init` becomes the new parent of child, and now handles `wait()` and cleanup.

Zombies happen when the child process terminates before the parent has called `wait()`. Child proces becomes a zombie process, and may fill up the process table.

Implementing `fork()`

`fork()`

1. Creates address space of child process
2. Allocates a new PID
3. Creates kernel process data structures, ie. PCB
4. Copies kernel environment of parent process, ie. priority
5. Initializes child process context, ie. PID, PPID, CPU_time=0
6. Copies memory regions from parent (Program, data, stack). Is a very expensive operation
7. Acquires shared resources like open files, current working directory.
8. Initializes hardware context for child process (registers, PC, pointers)
9. Child process ready to run, add to scheduler queue.

- `fork()` calls `clone()`, which is offered by Linux.

Process Scheduling

Concurrent processes means that multiple processes are running at the same time. This is achieved through either virtual or physical parallelism.

Processing environment

- Batch processing: No user interaction, no responsivity needed
- Interactive (Multiprogramming): User interaction, responsivity needed
- Real-time: Time-sensitive, must meet deadlines

Criterions for Scheduling

- Fairness: All proceses should have a fair share of CPU time, either on a per process or per user basis. Means no starvation.
- Utilization: All parts of the computing system should be utilized.
- Throughput: Number of processes completed per unit time
- CPU utilization: Percentage of time CPU is busy
- Predictability: Variation in response time
- Turnaround time: Total time taken, ie. end time - start time
- Response time: Time taken for a process to start responding to a request
- Wait time: Total time a process is ready but not running, ie. `turnaround time` - `burst time`

Scheduling policies

- Non-preemptive (cooperative): Process runs until it voluntarily gives up the CPU
- Preemptive: A process is given a fixed time quota to run, after which it is preempted and another process is run.

Batch Scheduling Algorithms

First-Come-First-Serve (FCFS)

- Simple, but can lead to convoy effect where a long process holds up short processes
- Guaranteed no starvation as number of tasks infront of x is strictly decreasing.

Shortest Job First (SJF)

- Guarantees minimal average waiting time, but can lead to starvation
- Need to know total CPU time in advance. Can be predicted by exponential averaging.
- $\text{pred}_{n+1} = \alpha \cdot \text{actual}_n + (1-\alpha) \cdot \text{pred}_n$

Shortest Remaining Time First (SRTF)

- Preemptive version of SJF

Interactive Scheduling Algorithms

RR

- Tasks run for a fixed time quantum, after which they join the back of a queue.
- Guarantees response time of $(n-1) \times q$ for n tasks.
- Timer interrupt is needed
- Bigger time quantums leads to better CPU utilization (less context switching), but worse response times.

Priority Scheduling

- Some processes are more important than others, we assign a priority value to all tasks, and run the highest priority task first.
- Can lead to starvation of low priority tasks. This can be tackled by aging, where the priority of a task decreases the longer it runs.
- Priority inversion can occur, where a high priority task is blocked by a low priority task.

Multilevel Feedback Queue

Basic rules

- If priority(A) > priority(B), A runs before B.
- If priority(A) = priority(B), A and B run in RR.

Priority rules

- New tasks → highest priority
- Tasks use up their time quantum → lower priority
- Tasks blocks before time quantum → maintain priority
- Minimizes both Response time for IO-bound tasks and Turnaround time for CPU-bound tasks.

Lottery Scheduling

- Each process is given a number of lottery tickets, and a random ticket is drawn. The process with the winning ticket runs. In the long run, a process with $x\%$ of the tickets will run $x\%$ of the time.
- Responsive (a newly created process can participate in lottery immediately)
- Good level of control. A process can be given more tickets to run more often, and can also disitribute tickets to child proceses. Each resource can also have its own set of tickets.

Inter-Process Communication

Shared Memory

A process may create a shared memory region, which other processes can attach to their own memory space. The processes can now communicate using this memory region.

Pros

- Only creating and attaching to shared memory involves OS, making it efficient
- Easy to use as the shared memory region behaves the same as normal memory space

Cons

- Access synchronization is needed
- Implementation is harder

Example code

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

// Create shared memory region
int shmid = shmget(IPC_PRIVATE, 40, IPC_CREAT
| 0600);

// Attach to shared memory region
int *shared_mem = shmat(shmid, NULL, 0);

// Read/Write to shared memory region
int x = shared_mem[0];
shared_mem[0] = 10;

// Detach and destroy shared memory region
shmdt((char*)shm);
shmctl(shmid, IPC_RMID, 0);
```

Message Passing

Processes can send and receive messages using system calls. The OS manages this, and the message is stored in kernel space.

1. Direct Communication

- Processses must name each other explicitly using the Unix domain socket.
- Only one link per pair of processes, and other process needs to be known.

2. Indirect Communication

- Messages are sent to and received from mailboxes/ports (message queue in Unix)
- One mailbox can be shared among multiple processes

Synchronization behaviours

- Blocking Primitives (synchronous)
  - Receive() : Receiver is blocked until message has arrived.
- Non-blocking Primitives (asynchronous)
  - Receive() : Receiver receives message if available, else some placeholder

Pros and Cons

+ Portable, can be easily implemented on different processing environments
+ Easier to synchronize than shared memory, as synchronization primitives can be used.
- Inefficient, as OS intervention is required and messages are copied to kernel space

Unix Pipes

A pipe forms a form of producer-consumer relationship, where writing is done in one pipe end and reading in the other. Data is streamed through memory and there are no file interactions. Unnamed pipes require a parent-child relationship between the processes, while named pipes require a filesystem lookup.

Pipe functions as a circular buffer with implicit synchronization. If the buffer is full, the writer is blocked. If the buffer is empty, the reader is blocked. Depending on Unix version, pipes may be half-duplex or full-duplex.

Also possible to attach a pipe to a file descriptor, redirecting input/output of the pipe to other programs. This can be done with `dup2()`.

Example code

```c
#define READ_END 0
#define WRITE_END 1

int main() {
  int pipe[2];
  char buffer[20];
  char *str = "Test message";

  pipe(pipe);

  if (fork() > 0) {     // Parent
    close(pipe[READ_END]);
    write(pipe[WRITE_END],
          str,
          strlen(str) + 1);
    close(pipe[WRITE_END]);
  } else {              // Child
    close(pipe[WRITE_END]);
    read(pipe[READ_END],
         buffer,
         sizeof(buffer));
    close(pipe[READ_END]);
    printf("Received: %s\n", buffer);
  }
}
```

Unix Signals

A signal is a asynchronous notification sent to a process. The signal recipient must handle this through either a default handler or a user-defined handler. Examples of signals are `kill`, `interrupt`, `stop`, `continue`.

Signals are sent to a process by either a the OS or another process, whereas interrupts are sent by hardware.

Example of a Signal handler

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void seg_handler(int signo) {
  if (signo == SIGSEGV) {
    printf("Caught SIGSEGV\n");
    exit(1);
  }
}

int main() {
  int *test = NULL;
  if (signal(SIGSEGV, seg_handler)
      == SIG_ERR)){
    printf("Handler registering failed\n");
  }
  *test = 10;
  return 0;
}
```

Threads

Processes are expensive, with significant cost in creation (Duplicating memory space, process context) and context switching (Saving and restoring process context).

A single processs can have multiple threads in a multithreaded process, where threads in the same process share

- Memory Context: Text, Data, Heap
- OS Context: PID, resources like page tables, fd tables

Threads retain certain unique information, like

- Thread ID
- Registers
- Stack

Pros

- Lightweight, require much less resources than multiple processes to managed
- Threads can share most resources of a process, no need for IPC mechanisms
- Multithreaded programs are more Responsive
- Threads can take advantage of multiple CPUs

Cons

- System call concurrency: problems can arise when system calls are made in parallel
- Process Behavior: Impact on process operations (`fork()` duplicating threads?)

Thread implementations

User Thread

Threads may be implemented as a user library, where the OS is unaware of the threads.

Pros

- Portability: Multithreaded programs can run on any OS

- Thread operations are just library calls
- Higher customizability: User can implement own scheduling algorithms

Cons
- Scheduling must be done at a process level. If a thread blocks, the whole process blocks.
- Cannot take advantage of multiple CPUs

Kernel Thread

Threads are implemented by the OS, and the OS is aware of the threads. Kernel mode maintains a thread table, and is able to schedule threads independently.

Pros
- Kernel can schedule on thread levels, more than 1 thread can run simultaneously on multiple CPUs
  - Blocking of one thread does not block the whole process

Cons
- Slower thread operations, as system calls are needed (TRAP calls are expensive)
- Less flexibility. Kernel threads will be used by all multithreaded programs, can be hard to balance many features and requirements

Hybrid Thread model

Both Kernel and User threads exist, where a number of user threads may be mapped to a number of kernel threads. This allows for greater flexibility, and the concurrency of a process can be controlled.

POSIX Threads

POSIX threads define a standard API for thread creation, synchronization and management. Implementation is not specified, and `pthread` can be implemeted as either user or kernel threads.
- `#include <pthread.h>` header file
- `gcc xxx.c -lpthread` to compile
- `pthread_t` is a data type for thread IDs
- `pthread_attr_t` is a data type for thread attributes

Creation Syntax

```
pthread_create(
    pthread_t *tid,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *args);
```

- `tid` is the thread ID
- `attr` is the thread attributes
- `start_routine` is a function pointer to the function to be executed by the thread
- `args` is the arguments to be passed to the function

Termination Syntax

```
int pthread_exit(void *retval);
```

- `retval` is the return value of the thread to whoever synchronizes with the thread
- Without `pthread_exit()`, the thread will terminate when the function returns.
- Without a return value, retval is not well defined

Synchronization

```
pthread_join(pthread_t tid, void **status)
```

- waits for the thread to terminate, and stores the return value in `status`

Synchronization

Race conditions occur when there is incorrect execution due to unsynchronized access to shared modifiable resources.

Critical Section

Properties of correct synchronization
- **Mutual Exclusion**: Only one process can be in the critical section at a time
- **Progress**: If no process is in the critical section, a waiting process should be granted access
- **Bounded Waiting**: After process p requests to enter the critical section, there is an upper bound on the number of times other processes can enter the critical section before p.
- **Independence**: Process not in the critical section should never block other processes

Symptoms of incorrect Synchronization
- **Deadlock**: All processes are blocked, waiting for each other to release resources
- **LiveLock**: Due to deadlock avoidance mechanisms, processes are not blocked, but are not making progress
- **Starvation**: A process is unable to progress due to other processes always being granted access to the critical section

Synchronization Mechanisms

Assembly Level implementations

Test-and-Set
- `TestAndSet $register, $memory_location`
- Atomic operation that sets a memory location to 1 and returns the previous value
- Uses busy waiting, wastes CPU cycles

```
void enter_critical(int *Lock) {
    while (TestAndSet(Lock) == 1);
}
```

```
void exit_critical(int *Lock) {
    *Lock = 0;
}
```

Intel x86 CPUs use the `xchg` instruction, which exchanges contents of two locations atomically. This can also be used to implement `TestAndSet`.

High Level Language implementations

Peterson's Algorithm

```
// Process 0
want[0] = 1;
turn = 1;
while (want[1] && turn == 1);
/* CRITICAL SECTION */
want[0] = 0;
```

```
// Process 1
want[1] = 1;
turn = 0;
while (want[0] && turn == 0);
/* CRITICAL SECTION */
want[1] = 0;
```

- Assumption is that writing to `turn` is atomic
- Employs busy waiting instead of entering a blocked state
- Too low level, higher-level constructs are desirable

Semaphores

A semaphore is an integer variable that can only be accessed through two atomic operations: `wait()` and `signal()`

```
// wait, p, down
wait(s) :
  if s <= 0 :
    block
  s--
```

```
// signal, v, up
signal(s) :
  s++
```

Properties

Since $S_{\text{initial}} \geq 0$, we have the following invariant

$$S_{\text{current}} = S_{\text{initial}} + \#\text{signals} - \#\text{waits}$$

If all processes call `wait()` before `signal()` when entering the critical section,

$$N_{\text{critical}} = \#\text{waits} - \#\text{signals}$$

Implementing General Semaphore with Binary S.

`count = x` where `x` is the initial value of the semaphore

```
mutex = Semaphore(1);
queue = Semaphore(0);
```

```
void general_wait() {
    wait(mutex);
    count--;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
void general_signal() {
    wait(mutex);
    count++;
    if (count <= 0) {
        signal(queue);
    } else {
        signal(mutex);
    }
}
```

Classical Synchronization problems

Producer Consumer Problem (Bounded Buffer)

Processes share a bounded buffer of size K, and producers add items to the buffer when it is not full, and consumers remove items when it is not empty.

Busy Waiting Solution

```
// Producer
while (true) {
    item = produce_item();

    while (!can_produce);
    wait(mutex);
    if (count < K) {
        buffer[in] = item;
        in = (in + 1) % K;
        count++;
        can_consume = true
    } else {
        can_produce = false;
    }
    signal(mutex);
}
```

```
// Consumer
while (true) {
    while (!can_consume);
    wait(mutex);
    if (count > 0) {
        item = buffer[out];
        out = (out + 1) % K;
        count--;
        can_produce = true;
    } else {
        can_consume = false;
    }
    signal(mutex);

    consume_item(item);
}
```

Blocking Solution

```
// Producer
while (true) {
    item = produce_item();

    wait(not_full);
    wait(mutex);
    buffer[in] = item;
    in = (in + 1) % K;
    count++;
    signal(mutex);
    signal(not_empty);
}
```

```
// Consumer
while (true) {
    wait(not_empty);
    wait(mutex);
    item = buffer[out];
    out = (out + 1) % K;
    count--;
    signal(mutex);
    signal(not_full);

    consume_item(item);
}
```

We can use 2 semaphores to represent the state of the buffer, `not_full = Semaphore(K)` and `not_empty = Semaphore(0)`. We also need a mutex semaphore to protect the buffer.

Readers-Writers Problem

Processes share a modifiable data structure, where readers can read simultaneously, but writers must have exclusive access.

```
// Writer
while (true) {
    wait(room_empty);
    write_data();
    signal(room_empty);
}
```

```
// Reader
while (true) {
    wait(mutex);
    readers++;
    if (readers == 1) {
        wait(room_empty);
    }
    signal(mutex);

    read_data();

    wait(mutex);
    readers--;
    if (readers == 0) {
        signal(room_empty);
    }
    signal(mutex);
}
```

This process hinges on making sure that the readers do not block each other and that we can accurately maintain the state of room empty given multiple readers. We use `mutex` to ensure that the readers synchronize among themselves, and `room_empty` to synchronize readers and writers together. Only the first reader to enter the room and the last reader to exit the room will update the `room_empty` semaphore.

Dining Philosophers

There are N philosophers seated around a circular table, with a single chopstick placed between each philosopher. A philosopher must have 2 chopsticks to eat.

Tanenbaum's Solution

```
// Philosopher
void philosopher(int i) {
    while(true) {
      think();
      take_chopsticks(i);
      eat();
      put_chopsticks(i);
    }
}
```

```
void take_chopsticks(int i)
{
    wait(mutex);
    state[i] = HUNGRY;
    safe_to_eat(i);
    signal(mutex);
    wait(s[i]);
}
```

```
void safe_to_eat(int i) {
    if (state[i] == HUNGRY
&&
        state[LEFT] !=
EATING &&
        state[RIGHT] !=
EATING) {
        state[i] = EATING;
        signal(s[i]);
    }
}
```

```
void put_chopsticks(int i) {
    wait(mutex);
    state[i] = THINKING;
    safe_to_eat(LEFT);
    safe_to_eat(RIGHT);
    signal(mutex);
}
```

Limited Eater Solution

If at most $N - 1$ philosophers are allowed to eat at the same time, we can guarantee that at least one philosopher will be able to eat, preventing deadlock.

```
void philosopher(int i) {
    while (true) {
        think();
        wait(seats);
        wait(chopstick[left]);
        wait(chopstick[right]);

        eat();

        signal(chopstick[left]);
        signal(chopstick[right]);
        signal(seats);
    }
}
```

Memory management

The OS provides an illusion that a process owns the entire memory space. The OS also isolates the memory space of each process from each other by providing different physical addresses.

Memory Hierarchy

CPU registers → Cache Memory (SRAM) →
Main Memory (DRAM) → Disk Storage

Physical Memory storage

RAM can be treated as an array of bytes, where each byte has a unique index known as a physical address.

Types of data

1. Transient Memory: Data that is only needed for a short period of time, like function arguments and local variables
2. Persistent Memory: Data that needs to be stored for a long time, like global variables and heap memory

Memory Abstraction

Without memory abstraction, the address in programs would be physical addresses, and no external conversions or mappings are required. The addresses are also fixed during compile time.

This is not viable as two processes may assume memory starrts and 0, and may overwrite each other's data, and the memory space is not protected.

Static Address Relocation

The OS recalculates all memory references when the process is loaded into memory, adding some offset to all addresses. Cons:

- Slow loading time for processes
- Difficult to distinguish memory references from normal integer constant, or memory references in registers

Dynamic Address Relocation (Base and Limit)

Base register stores the starting address of the process, and a limit register stores the size. All memory references are relative to the base register, and access is checked against the limit to protect memory space integrity. Pros:

- Fast loading time
- Fast address translation as it is handled by hardware (MMU)

Contiguous Memory management

A process must be in memory during execution. To support multitasking, the OS must allow multiple processes in the physical memory at the same time.

Memory Partitioning

Fixed Partitioning

Memory is divided into fixed partitions, and processes are loaded into these partitions. Internal fragmentation occurs.

- Easy to manage, fast to allocate (every partition is the same)
- Partition size needs to be large enough to contain the largest process

Variable Partitioning

Partition is created based on the actual sizes of processes. External fragmentation occurs.

- Flexible, less wasted memory and no chance of internal fragmentation
- More complex to manage, slower allocation to find space

Variable partitioning allocation Algorithms

- **First Fit**: Allocate the first partition that is big enough
- **Best Fit**: Allocate the smallest partition that is big enough
- **Worst Fit**: Allocate the largest partition
- **Next Fit**: Allocate the next partition that is big enough, starting from last allocation
- When a occupied partition is freed, we merge with adjacent holes if possible
- Compaction can also be used to move occupied partitions around to consolidate holes, but is expensive

Implementations

Partition information may be managed using a linked list, where each node contains the size of the partition and a pointer to the next partition.

A buddy system can also be used, where partitions are split into halves until the smallest partition is reached. We maintain an array A of linked lists, where A[i] contains all partitions of size 2^i.

- To allocate a block of size N, we find the smallest i such that $2^i \geq N$, and allocate from A[i]. If none exist, we split larger blocks until we reach the desired size.
- To free a block, we check if the buddy is free, and merge if possible.
- Two blocks B and C are buddies of size s if the sth bit of B and C are complements, and they share the same prefix.

Note that the buddy system can suffer from both internal and external fragmentation, though internal fragmentation is limited to $< 50\%$.

Disjoint Memory Schemes

Paging

Physical memory is divided into fixed-size regions known as **physical frames**, and the logical memory is also divided into regions of same sizes known as **logical pages**. At execution time, the pages of a process are loaded into any available memory frame.

We need a lookup table to map logical to physical memory. The program code uses logical memory, and to locate this in physical memory, we need to know the frame number and the offset within the frame.

With our page size 2^n and m bits of logical address `LA`, page number p = most significant $m - n$ bits, and offset d = remaining n bits. Then physical address `PA` = $f \times 2^n + d$.

- Paging removes external fragmentation, but introduces internal fragmentation as logical memory space may not be multiple of page size.
- Paging also allows for clear separation of logical and physical memory, and allows for easy sharing of pages between processes.

Implementation

A pure-software implementation may be through the OS storing page table information with other process information in the PCB. The memory context of a process is now the page table. However, this is slow as now it requires two memory accesses for every memory reference - 1st to the page table to get frame number, 2nd is to access the actual memory item.

Hardware support for paging is available in the **Translation Look-Aside Buffer**. the TLB acts as a cache of some page table entries. The TLB is faster as it is implemented in sRAM, and decreases the number of lookups relative to a multi-level page table.

1. Use page number to search TLB for frame number.
2. If TLB-Hit, use frame number and offset to access memory.
3. If TLB-Miss, Trap call to OS.
   - Access full page table of process in process PCB
   - Check if valid bit set. If No, raise segmentation fault
   - Update TLB with PTE
   - Return from Trap

The TLB is part of the hardware context of a process. When a context switch occurs, the TLB entries are flushed so the next process can use the TLB. We can also have software managed TLB, where we must check in the PCB of the process for a page table.

Memory Protection in Paging

Access-Right Bits

Each page table entry has bits that indicate `read` - `write` - `execute` for groups of `owner` - `group` - `universe`. Memory access is first checked against the access right bits.

Valid bits

Attached to each page table entry is a valid bit which indicates whether the page is valid for the process to access. Some pages are out-of-range for a particular process, and OS will catch these accesses using these bits.

Page sharing

Pages can be shared between processes, and the same physical frame number is used in the page table of both processes. This then calls for **Copy-On-Write**, where the OS only copies the page when one of the processes tries to write to the page.

Copy-on-Write

Copy on write causes the `write` access bit to be set to `0` in the page table for both processes. When OS detects a memory violation, copy on write is performed.

1. Locate a free frame
2. Copy the original frame to the free frame
3. Update the page table of the process to point to the new frame
4. Set the `write` access bit to `1`
5. Perform write

Segmentation Schemes

In a process, there are different memory regions like code, data, stack, heap. Different regions behave differently (ie. growing) and it is hard to place different regions in a contiguous memory space and allow them to grow and shrink freely, in addition to checking for in-range memory access.

The logical memory of a process is now a collection of segments. Each segment has a name and limit, and is mapped to a contiguous physical memory region with a base address and limit. With logical address < `segment_id`, `offset` >, we can lookup the segment table with key `segment_id` and value (`base`, `limit`) to obtain the physical address `base` + `offset`. Segmentation is susceptible to external fragmentation.

Hardware Support for Segmentation

The translations for segmentation is done in the hardware in the form of a segment table.

1. Address is split into segment number and offset
2. Segment number is used to lookup the segment table to get the base address and limit.
3. The offset is checked against the limit to ensure it is in range.
4. The physical address is then calculated as `base` + `offset`.

Segmentation with Paging

Each segment is composed of several pages instead of a contiguous memory region, ie. each segment has a page table. The memory of a process is now a collection of segments, each with a number of pages which all fit into physical memory frames.

1. The logical address is split into `segment_number`, `page_number`, and `offset`.
2. Lookup `segment_number` in the segment table to get the *page table base address* and limit.
3. Check the offset against the limit to if it is in range. If no, raise a segmentation fault.
4. Lookup `page_number` in the page table to get the frame number.
5. Calculate the physical address as `frame_number` + `offset`.

Virtual Memory management

Secondary storage has a much larger capacity compared to physical memory, and with virtual memory we can use secondary storage as an extension of physical memory.

There are two page types, memory resident pages in physical memory and non-resident pages in secondary storage. The CPU can only access memory resident pages, and otherwise a page fault occurs.

Accessing a page

0. Check TLB
1. Check page table if page is memory resident. If yes, access location.
2. If not, raise a page fault, and we make a Trap call to OS.
3. Locate page in secondary storage
4. Locate victim page to replace with replacement algorithm
5. Load page into a physical memory frame
6. Update page table entry
7. Update TLB
8. Return from Trap
9. Go to step 1, retry

Thrashing occurs when memory access results in many page faults, but the chance of this happening is low due to locality of reference.

- **Temporal locality**: Recently accessed pages are likely to be accessed again. This "amortizes" the cost of loading the page into memory.
- **Spatial locality**: Pages near recently accessed pages are likely to be accessed, and later accesses to nearby locations will not cause page faults.

Key aspects of Virtual Memory

1. Managing a huge page table efficiently
2. Handling page faults
3. Deciding how to distribute frames among processes

Page Table structures

Page Table Entry

PTE sizes are influenced by size of physical memory as it stores the frame number.

Direct Paging

All entries are in a single table, and the page number is used as an index to the table. With a logical memory that fits 2^p pages, the page table has 2^p page entries, each which contain a physical frame number, and additional information bits like valid bits and access rights.

$$p = \log_2(\text{Logical Memory Space}) - \log_2(\text{Page Size})$$
$$\text{Size of Page table} = 2^p \times \text{Size of Page Table Entry}$$

Multilevel Paging

We can split page table into smaller page tables. With the original page table of 2^p entries, we can split this into 2^m page tables of 2^{p-m} entries. The first m bits of the logical address are used to index the first level page table, and the next $p - m$ bits are used to index the second level page table. The second level page table entry contains the physical frame number.

This reduces the size of the page table as only the necessary page tables are loaded into memory, and the rest do not need to be allocated.

Inverted Page Table

In a normal page table, the entries are ordered by page number. With M processes, there are M independent page tables, but only N physical memory frames can be occupied. Since $N << M$, there is a huge amount of wasted space.

In an inverted page table, have a mapping of `frame number` to < `pid` , `page number` >. This reduces the size of the page table to N entries, but the lookup time is now $O(N)$.

Page Replacement Algorithms

When a page fault occurs and there is no free memory frame, we must replace a page.

Memory Access Time

With probability of page fault p,

$$T_{\text{access}} = (1-p) \times T_{\text{mem}} + p \times T_{\text{page\_fault}}$$

Optimal Page Replacement

In an imaginary scenario, where we have future knowledge of memory references, we replace the page that will not be used again for the longest period of time, and this guarantees the lowest page fault rate.

FIFO

We evict the page that has been in memory the longest, easily implemented with a queue of page numbers.

FIFO suffers from **Belady's Anomaly**, where increasing the number of frames can lead to an increase in page faults. This is because FIFO does not exploit temporal locality.

LRU

We evict the page that has not been used for the longest period of time. This directly exploits temporal locality, as we expect the least recently used page to be the least likely to be used again. This attempts to approximate OPT, and does not suffer from Belady's Anomaly.

Implementations

- **Counter**: Each PTE has a timestamp, and the page with the smallest timestamp is evicted. We need to search through all pages, and timestamp is forever increasing and may overflow.
- **Stack**: When page is referenced, we remove and push to top of stack. The page at the bottom of the stack is the least recently used. This is not a pure stack.

2nd Chance Page Replacement

Each PTE has a reference bit, set to 1 when a page is referenced. When a page has reference bit 1 and is about to be replaced, we set the reference bit to 0 and move it to the back of the queue. This can be implemented with a circular queue, ie. an array with a pointer to the oldest victim page. When all reference bits == 1, this becomes the FIFO algorithm.

Frame allocation

With N physical memory frames, M proccesses, how do we distribute the frames among the processes?

- Equal Allocation: Each process gets $\frac{N}{M}$ frames, may not be fair as some processes may need more memory.
- Proportional Allocation: Allocate frames based on the size of the process, ie. $\frac{N_{\text{process}}}{N_{\text{total}}} \times N$ frames.

Replacement policies

Local page replacement occurs when victim pages are selected among pages of the process that causes the page fault.

+ Frames allocated to process remains constant, resulting in stable performance between multiple runs.
- If not enoughb frames are allocated, performance is poor.

Global page replacement occurs when victim pages are selected among all pages in memory.
+ Allows self-adjustment between processes, where a process can steal frames from another.
- Badly behaving processes can affect others
- Frames allocated to a process can be different on each run

Insufficient physical frames result in thrashing, and there is heavy I/O to bring non-resident pages into RAM.

- If global replacement is used, a thrashing process may steal pages from other processes, causing them to start thrashing. This is cascading thrashing.
- If local replacement is used, the thrashing is limited to one process, but it can hod the I/O and degrade performance of other processes.

Working Set model

The set of pages referenced by a process is relatively constant in a period of time due to locality. Referring to instructions sequentially has spatial locality, and running a loop has temporal locality.

As such, we define a working set window Δ, and determine the active pages in the interval at time t as the working set $W(t, \Delta)$. We allocate enough frames for pages in $W(t, \Delta)$ to reduce page faults.

If Δ is too small, we may miss pages in the current locality, but if Δ is too big, we risk containing pages from multiple localities. Additionally, OS may consider the working set of all processes to allocate frames. With too high of a Δ, more processes may be blocked from running on CPU, resulting in low page fault rates but low CPU utilization.

File Systems

The OS provides an abstraction of a File as a single contiguous logical entity, and protects the file by ensuring that they can only be opened through system calls, enabling the OS to enforce access control. Physical memory is volatile, so external storage is needed to store persistent data. File systems should be

- Self contained: All information needed to manage the file system is contained within the file system
- Persistent
- Efficient: Fast management, minimum overhead for bookkeeping

File System Abstractions

File

A file is a abstraction of some data, containing data and metadata (file attributes).

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file

OSes commonly support a number of file types, each with their own set of operations. Common file types are regular files, directories and special files.

Regular files are further divided into 2 types, text files and binary files. Binary files (executables, Java class file, `.pdf` / `.jpg` / ...) have predefined internal structures that are processed by specific programs.

File types may be distinguished in two Ways

1. File extension: Change of extension changes the file type
2. Embedded information: Magic Number in Unix at the beginning of the file

File Protection

Permission Bits

We classify users into 3 classes: Owners, Group (set of users), Universe. We then define RWX permissions for each 3 classes, visible though `ls -l`.

Access Control List

In Unix, ACL can be minimal (same as permission bits) or extended, where we can define permissions for each specific user/group. This is more flexible than permission bits, but is more complex to manage with more additional information associated with each file.

File Data Structure

- **Array of bytes**: No interpretation of data, just a sequence of bytes
- **Fixed-length records**: Data is an array of records which may shrink or grow. Random access is possible with offset.
- **Variable-length records**: Records are of variable length, and a pointer to the next record is stored in the current record. This is more flexible, but slower to access.

Access Methods

1. **Sequential access**: Data read in order, cannot skip but can rewind
2. **Random Access**: Data can be read in any order.
   - `Read(offset)` : Read data at offset
   - `Seek(offset)` : Move pointer to offset
3. **Direct Access**: Used for files containing fixed-length records, it provides random access to any record.

File Operations

- **Create**
- **Open**: File is opened for reading/writing, and a file descriptor is returned
- **Read/Write**: Data is read/written from/to file from current position

- **Seek**: Move file pointer to a specific position
- **Truncate**: Remove data between specified position to end of file

The OS provides file operations as system calls, providing protection, concurrency and efficient access.
For each open file, it stores several information:
- **File pointer**: Current position in file
- **Disk Location**: Actual file location on disk
- **Open Count**: Number of processes that have opened the file

Process-File Organization
- **System-wide table**: One entry per unique file
- **Per-process table**: One entry per file opened by the process

In Unix, each process has a per-process file descriptor table, and a system-wide file table. The file descriptor table contains pointers to the system-wide file table, and the system-wide file table contains information about the file, like the file pointer and disk location.

Processes may share a file through two different file descriptors so that I/O can occur at independent offsets. Processes may also share the same file descriptor, as in the case where `fork()` is called.

Directories
Directories provide a logical grouping of files. They may be structured in several ways
- **Single Level**
- **Tree**
  - Directories can be recursively embedded in other directories, and may be referred through absolute or relative paths.
- **DAG**
  - A DAG may be formed from the Tree structure when a file is shared, and a file "appears" in multiple directories.
  - This may be implemented in Unix in the form of hard or symbolic links.
- **General graph**
  - A general graph may be formed when a symbolic link is allowed to link to a directory (allowed in Unix), forming a loop.
  - Undesirable, not easy to traverse and may be difficult to determine when to remove a directory.

Hard Link
A hard link is a directory entry that points to a target file inode. A and B have separate pointers point to the actual file f in the disk. This incurs low overhead, as only pointers are added into the directory. However, this may cause issues if the file is deleted from a directory.

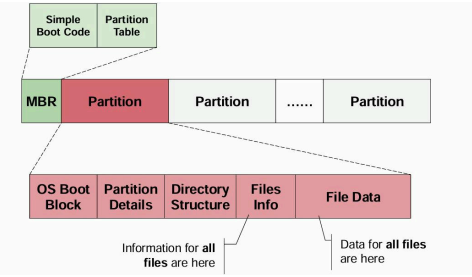
Symbolic Link
A symbolic link is a inode that stores the path to target file as its data. B creates a special link file that contains the path to f. When this file is accessed, we find out where f is before accessing it. This solves the deletion problem, as the link file being deleted does not affect the original file, and the original file being deleted does not affect the link file. However, this may incur higher overhead as the link file takes up actual disk space.

Directory Permission Settings
A directory is just a file which contains a list of directory entries (of files and subdirectories). Directory permissions are not the same as file permissions, and have the following meanings:

- Read bit: Ability to read the list of directory entries
- Write bit: Ability to change the list of directory entries
- Execute bit: Ability to use this directory as your working directory
  - To allow others to access a file, they only need the execute bit.

File Systems Implementations
Storage devices may be treated as a 1D array of logical blocks, which are the smallest accessible units of storage. Blocks are mapped into disk sectors, which is hardware dependent.

Disk Organization
The Master Boot Record sits at sector 0 and contains the boot loader and partition table. Each partition contains an independent file system.



Implementing a File
A file is a sequence of logical blocks. A good file implementation should keep track of logical blocks, allow for efficient access and utilize disk space effectively. Additionally, internal fragmentation in the last logical block should be minimized.

Contiguous Allocation
We may allocate consecutive disk blocks to a file.

- Overhead per file: Starting block number + length of file/ blocks.
- **Pros**:
  - Simple: each block only needs starting block number + length
  - Fast access: Only need to seek first block, sequential access is fast
- **Cons**:
  - External fragmentation: Files may be scattered across disk
  - Need to know file size beforehand

Linked List
The data for a file may be scattered across the disk, and each block contains a pointer to the next block.

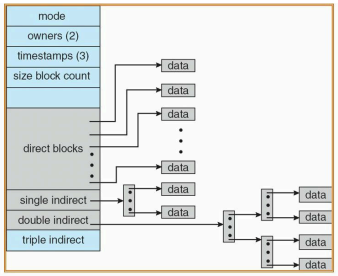
- Overhead per file: Each block needs to store a pointer to the next block. We also need a pointer to the first and last block in the file.
- **Pros**:
  - No external fragmentation
- **Cons**:
  - Poor Random access: Need to traverse linked list
  - Need to store pointer in each block, introduces point of failure

To improve on this, we can use a File Allocation Table (FAT). The FAT is in memory and contains a table of pointers to the next block. To find a particular file, we first find the starting block of a given file in a hash table, and then for each block b, we find the next block by looking up `FAT[b]`.
- Overhead per file: A pointer (in memory) for each block, a pointer to the first block per file.
- **Pros**:
  - Faster Random access as the LL traversal now takes place in memory
- **Cons**:
  - Need to store FAT in memory, which may be large

Another LL implementation is Indexed Allocation. Each file has an index block in disk that contains a list of pointers to the actual data blocks.
- **Pros**:
  - Faster direct access
  - Smaller memory overhead, only index block of open file needs to be in memory
- **Cons**:
  - Limited file size, as the index block has a fixed size
  - Introduces a extra block of overhead per file

There are several variations. A Linked scheme may be used to keep a linked list of index blocks, s.t. the last element of the index block is a pointer to the next index block. Alternatively, a multilevel index may be used where the first level index block points to a second level index block.

Unix Indexed Node (iNode)
Unix uses a combination of direct indexing and multi-level indexing schemes. It has 12 direct blocks that point to disk blocks directly, and singular single, double and triuple indirect blocks. This provides a combination of efficiency for small files and flexibility for larger file sizes.



Free Space Management
For each partition, to perform file allocation, we need to know which disk block is free. This is done through a free space list. On allocation, we remove the free disk block from free space list, and on deallocation, we add the disk block back to the free space list.

Bitmap
- **Pros**:
  - Simple manipulation, can easily find first free block and n -consecutive free blocks through bit level operations.
- **Cons**:
  - Need to store in memory for efficiency reasons, can be large

Linked List
We maintain a LL of free blocks lists, where each block contains a table of some free blocks and a pointer to the next free block list.
- **Pros**:
  - Easy to locate free blocks
  - Only first pointer is needed in memory, others can be cached
- **Cons**:
  - Has high overhead, but can be mitigated by storing the free block list in free blocks

Implementing Directories
Directories are important to keep tracks of the files (and their metadata) in a file system.

Linear List
Directories are stored as a linear list of entries, where each entry contains the file name, metadata and the iNode number. When we locate a file, we must traverse the entire list, which may be inefficient for large directories. It may help to use caches to remember the latest few searches, as users tend to move up and down a singular path.

Hash Table
Directories are stored as a hash table with chaining. This provides fast lookup, but requires a good hash function for efficiency.

Storing File information
File information consists of file names, metadata and disk block information. Directories may choose to store these information in the iNode, or in a separate data block.

File Systems Operations
At runtime, with user interactions the OS maintains run-time information in memory. These include:
- System-wide open-file table
- Per-process open-file table
- Buffers for disk blocks read/write operations

System-wide file table
Each entry contains:
- fd info
- file offset
- access mode
- reference count
- inode pointer

The per-process file table contains pointers to fds in the system-wide file table.

Create

To create a file, we

1. Use full pathname to locate parent directory
   - Check if file already exists
2. Use free space list to allocate disk blocks
3. Add entry to parent directory

Open

1. Search system-wide open-file table for file
   - If existing entry found, return file descriptor
2. Use pathname to locate file
3. Load file information into a new entry in system-wide table
4. Create a new entry in per-process table that points to the system-wide table entry
5. Return file descriptor

Unix File Operations

With the relevant header files

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

We can perform several file operations. An opened file has an identifier known as a file descriptor (`int`), and this file may be accessed on a byte-by-byte basis.

`open()`

```c
int open(char *path, int flags)
```

- Returns:
  - -1: Failed to open file
  - ≥ 0: File descriptor, a unique index for opened file
- Arguments:
  - `path` : Path to file
  - `flags` : `read` and/or `write` , truncation, append, create ...

```c
// Read-only
int fd = open("file.txt", O_RDONLY);
// Read-write, create if not exist
int fd = open("file.txt", O_RDWR | O_CREAT);
```

Some default file descriptors are `STDIN(0)` , `STDOUT(1)` , `STDERR(2)` .

`read()`

```c
int read(int fd, void *buffer, size_t count)
```

- Reads up to `count` bytes from file descriptor `fd` into `buffer`
- `read()` is sequential and begins from file pointer
- Returns:
  - Number of bytes read
- Arguments:
  - `fd` : (Opened) File descriptor
  - `buffer` : Buffer to store data

`write()`

```c
int write(int fd, void *buffer, size_t count)
```

- Writes up to `count` bytes from `buffer` to file descriptor `fd`
- `write()` is sequential and begins from file pointer. It can increase file size beyond `EOF` , appending new data.
- Returns:
  - -1: Error (Exceed file size limit, disk space etc.)
  - ≥ 0: Number of bytes written
- Arguments:
  - `fd` : (Opened) File descriptor
  - `buffer` : Buffer to write data from

`lseek()`

```c
off_t lseek(int fd, off_t offset, int whence)
```

- Moves file pointer of file descriptor `fd` to `offset` bytes from `whence`
- Return:
  - -1: Error
  - ≥ 0: New file pointer position
- Arguments:
  - `fd` : (Opened) File descriptor
  - `offset` : Number of bytes to move
  - `whence` : `SEEK_SET` , `SEEK_CUR` , `SEEK_END` , the point of reference from which we interpret the `offset`

`close()`

```c
int close(int fd)
```

- `fd` can no longer be used, kernel can remove associated data structures. By default process termination automatically closes all open files.
- Return
  - -1: Error
  - 0: Success

`dup2()`

```c
dup2(fd, STDOUT_FILENO)
```

- This makes stdout point to fd such that all writes to stdout are now redirected to file.